B-trees

Andreas Kaltenbrunner, Lefteris Kellis & Dani Martí

What are B-trees?

- B-trees are **balanced search trees**: height $= O\big(\log(n)\big)$ for the worst case.
- They were designed to work well on **Direct Access secondary storage devices** (magnetic disks).
- Similar to **red-black** trees, but show **better performance on disk I/O operations**.
- B-trees (and variants like **B+** and **B*** trees) are widely used in **database systems**.

Motivation

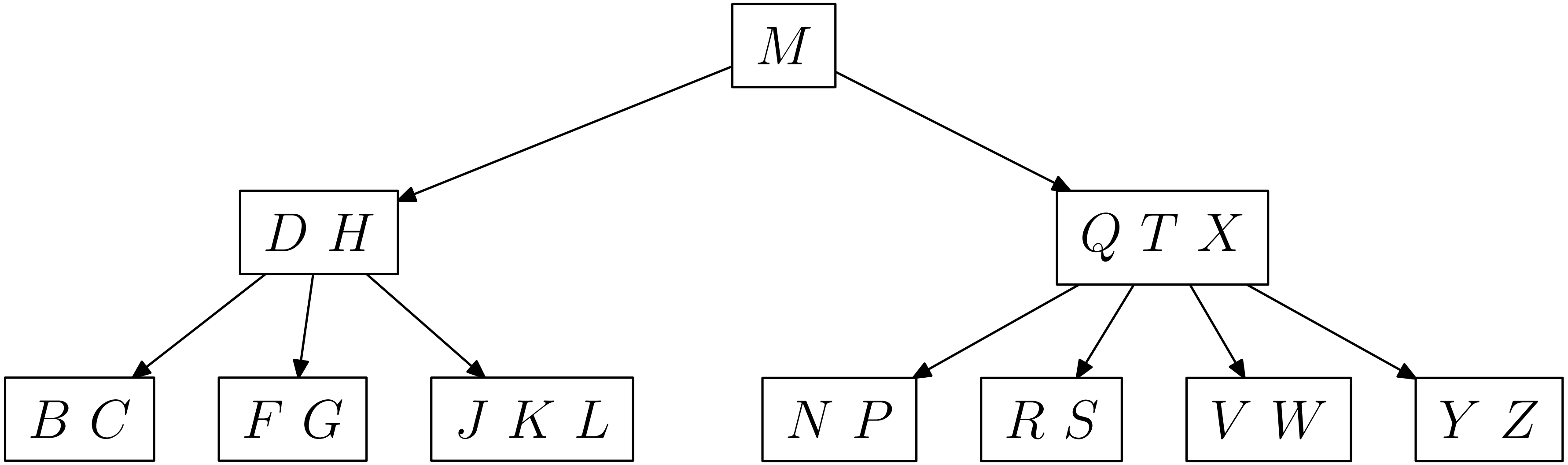
Data structures on secondary storage:

- Memory capacity in a computer system consists broadly on 2 parts:
  1. **Primary memory**: uses memory chips.
  2. **Secondary storage**: based on magnetic disks.
- Magnetic disks are **cheaper** and have **higher capacity**.
- But they are much slower because they have moving parts.

B-trees try to read **as much information as possible** in **every disk access** operation.

An example

The 21 english consonants as keys of a B-tree:



- Every internal node x containing $n[x]$ keys has $n[x] + 1$ children.
- All leaves are at the **same depth** in the tree.

B-tree: definition

A **B-tree** T is a rooted tree (with root $\mathrm{root}[T]$) with properties:

- Every node x has **four** fields:

  1. The number of keys currently stored in node x, $n[x]$.

  2. The $n[x]$ keys themselves, stored in nondecreasing order:

$$\mathrm{key}_1[x] \leq \mathrm{key}_2[x] \leq \cdots \leq \mathrm{key}_{n[x]}[x]\,.$$

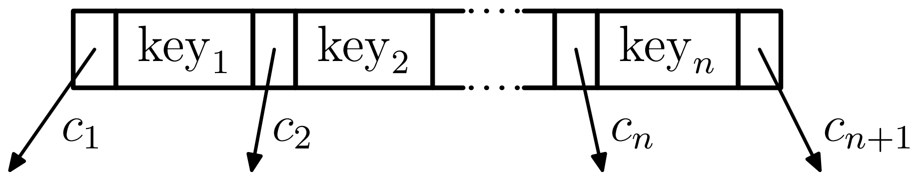
  3. A boolean value,

$$\mathrm{leaf}[x] = \begin{cases} \text{True} & \text{if } x \text{ is a leaf}\,, \\ \text{False} & \text{if } x \text{ is an internal node}\,. \end{cases}$$

  4. $n[x]+1$ pointers, $c_1[x]$, $c_2[x]$, $\ldots$, $c_{n[x]+1}[x]$ to its children.

  (As leaf nodes have no children their c_i are undefined).

- Representing pointers and keys in a node:



B-tree: definition (II)

Properties (cont):

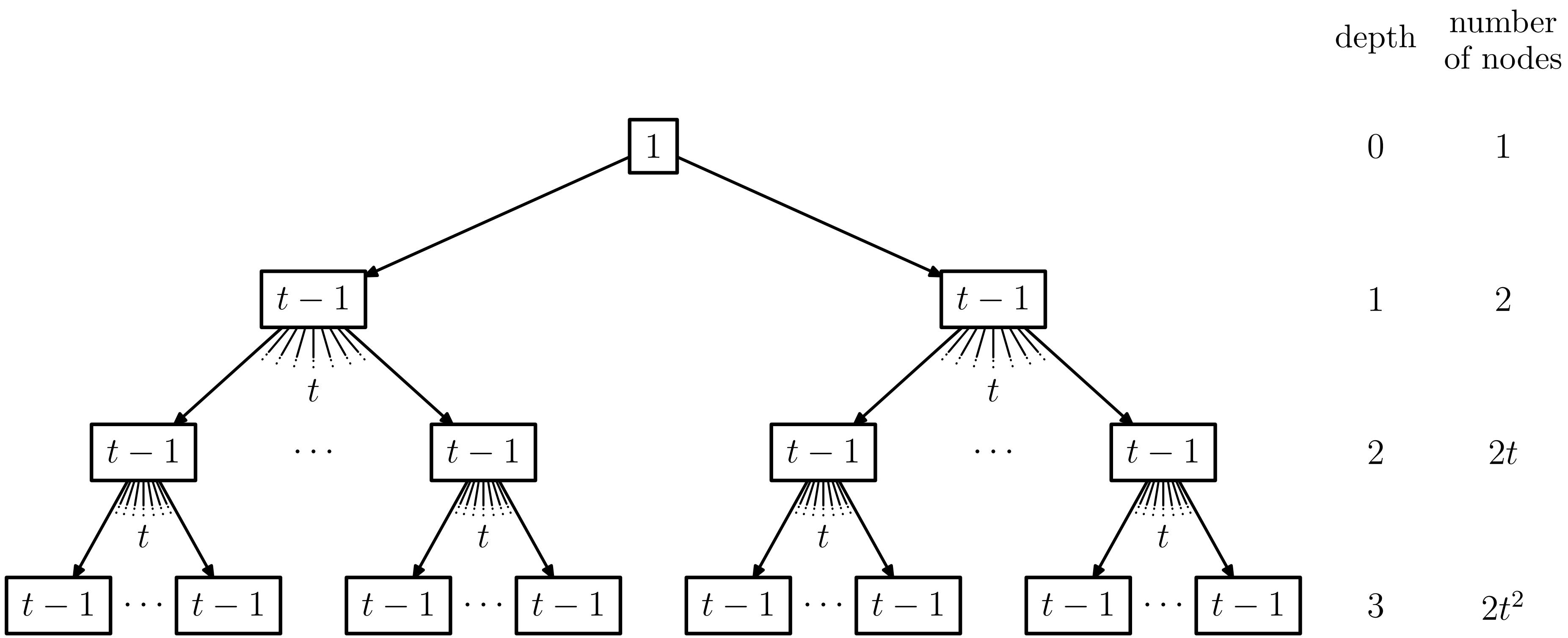
- The keys $\text{key}_i[x]$ separate the ranges of keys stored in each subtree: if k_i is **any** key stored in the subtree with root $c_i[x]$, then:

$$k_1 \leq \text{key}_1[x] \leq k_2 \leq \text{key}_2[x] \leq \ldots \leq \text{key}_{n[x]} \leq k_{n[x]+1} \,.$$

- All leaves have the **same height**, which is the tree's height h.
- There are upper on lower bounds on the number of keys on a node.
  To specify these bounds we use a fixed integer $t \geq 2$, the **minimum degree** of the B-tree:
  - **lower bound**: every node other than root must have at least $\boxed{t-1}$ keys $\Longrightarrow$ At least $\boxed{t}$ children.
  - **upper bound**: every node can contain at most $\boxed{2t-1}$ keys $\Longrightarrow$ every internal node has at most $\boxed{2t}$ children.

The height of a B-tree (I)

Example (worst-case): A B-tree of height 3 containing a **minimum** possible number of keys.



Inside each node x, we show the number of keys $n[x]$ contained.

The height of a B-tree (II)

- Number of **disk accesses** proportional to the **height** of the B-tree.
- The **worst-case height** of a B-tree is

$$h \leq \log_t \frac{n+1}{2} \sim O(\log_t n)\,.$$

- **Main advantadge** of B-trees compared to red-black trees:

  The base of the logarithm, t, can be much larger.

  $\Longrightarrow$ B-trees save a factor $\sim \log t$ over red-black trees in the number of nodes examined in tree operations.

  $\Longrightarrow$ **Number of disk accesses substantially reduced**.

Basic operations on B-trees

Details of the following operations:

- B-Tree-Search
- B-Tree-Create
- B-Tree-Insert
- B-Tree-Delete

Conventions:

- Root of B-tree is always in main memory (Disk-Read on the root is never required)
- Any node pased as parameter must have had a Disk-Read operation performed on them.

Procedures presented are all **top down algorithms** (no need to back up) starting at the root of the tree.

Searching a B-tree (I)

2 inputs: x, **pointer** to the root node of a subtree,
k, a **key** to be searched in that subtree.

function B-TREE-SEARCH(x, k) **returns** (y, i) such that $\text{key}_i[y] = k$ or NIL
  $i \leftarrow 1$
  while $i \leq n[x]$ and $k > \text{key}_i[x]$
    do $i \leftarrow i + 1$
  if $i \leq n[x]$ and $k = \text{key}_i[x]$
    then return (x, i)
  if leaf$[x]$
    then return NIL
    else DISK-READ$(c_i[x])$
      return B-TREE-SEARCH$(c_i[x], k)$

At each internal node x we make an $(n[x] + 1)$-way branching decision.

Searching a B-tree (II)

- Number of disk pages accessed by B-TREE-SEARCH

$$\Theta(h) = \Theta(\log_t n)$$

- time of **while** loop within each node is $O(t)$ therefore the total CPU time

$$O(th) = O(t \log_t n)$$

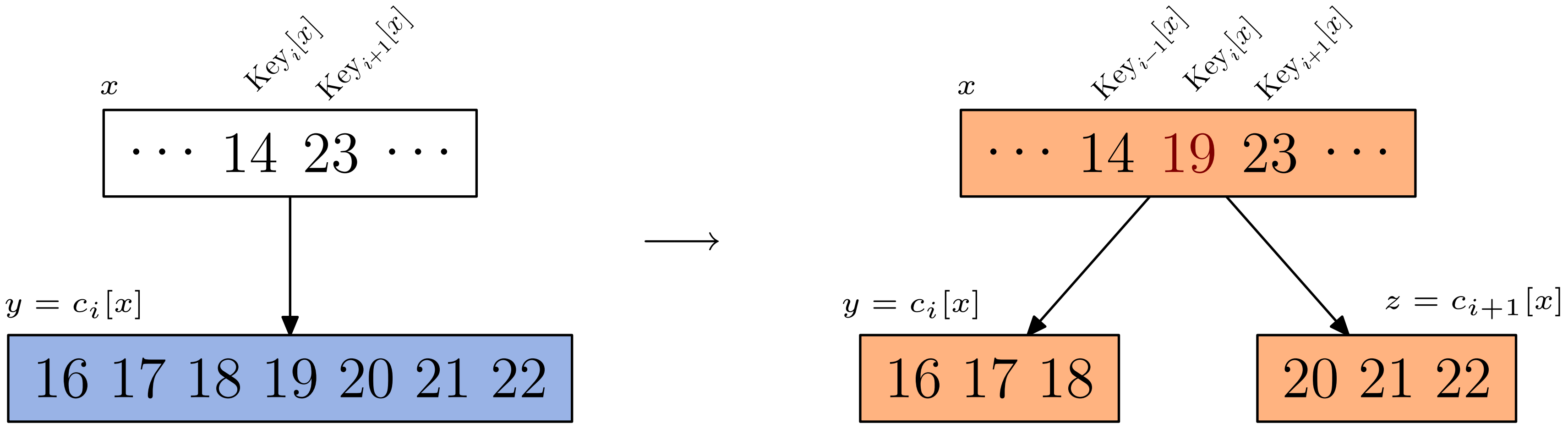
Creating an empty B-tree

```
B-TREE-CREATE(T)
    x ← ALLOCATE-NODE()
    leaf[x] ← TRUE
    n[x] ← 0
    DISK-WRITE(x)
    root[T] ← x
```

- ALLOCATE-NODE() allocates one disk page to be used as a new node
- requires $O(1)$ disk operations an $O(1)$ CPU time

Splitting a node in a B-tree (I)

- Inserting a key into a B-tree is more complicated than in binary search tree.
- **Splitting** of a full node y ($2t - 1$ keys) fundamental operation during insertion.
- Splitting around **median key** $\mathrm{key}_t[y]$ into 2 nodes.
- Median key moves up into y's parent (which has to be **nonfull**).
- If y is root node tree height grows by 1.



Splitting a node in a B-tree (II)

3 inputs: x, a **nonfull** internal node,
i, an index,
y, a node such that $y = c_i[x]$ is a **full** child of x.

B-Tree-Split-Child(x, i, y)
    $z \leftarrow$ Allocate-Node$()$
    $\text{leaf}[z] \leftarrow \text{leaf}[y]$
    $n[z] \leftarrow t - 1$
    for $j \leftarrow 1$ **to** $t - 1$
        do $\text{key}_j[z] \leftarrow \text{key}_{j+t}[y]$
    if not $\text{leaf}[y]$
        then for $j \leftarrow 1$ **to** t
            do $c_j[z] \leftarrow c_{j+t}[y]$
    $n[y] \leftarrow t - 1$

    for $j \leftarrow n[x] + 1$ **downto** $i + 1$
        do $c_{j+1}[x] \leftarrow c_j[x]$
    $c_{i+1}[x] \leftarrow z$
    for $j \leftarrow n[x]$ **downto** i
        do $\text{key}_{j+1}[x] \leftarrow \text{key}_j[x]$
    $\text{key}_i[x] \leftarrow \text{key}_t[y]$
    $n[x] \leftarrow n[x] + 1$
    Disk-Write(y)
    Disk-Write(z)
    Disk-Write(x)

CPU time used by B-Tree-Split-Child is $\Theta(t)$ due to the loops

Inserting a key into a B-tree (I)

- The key is always inserted in a leaf node
- Inserting is done in a single pass down the tree
- Requires $O(h) = O(\log_t n)$ disk accesses
- Requires $O(th) = O(t \log_t n)$ CPU time
- Uses $\textsc{B-Tree-Split-Child}$ to guarantee that recursion never descends to a full node

Inserting a key into a B-tree (II)

2 inputs: T, the root node,
k, key to insert.

```
B-TREE-INSERT(T, k)
  r ← root[T]
  if n[r] = 2t − 1
     then s ← ALLOCATE-NODE()
          root[T] ← s
          leaf[s] ← FALSE
          n[s] ← 0
          c_1[s] ← r
          B-TREE-SPLIT-CHILD(s,1,r)
          B-TREE-INSERT-NONFULL(s,k)
     else B-TREE-INSERT-NONFULL(r,k)
```

Uses B-TREE-INSERT-NONFULL to insert key k into nonfull node x

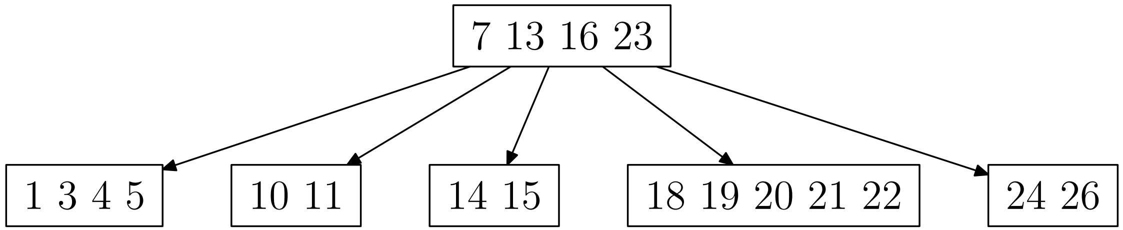
Inserting a key into a nonfull node of a B-tree

```
B-TREE-INSERT-NONFULL(x, k)
  i ← n[x]
  if leaf[x]
    then while i ≥ 1 and k < key_i[x]
             do key_{i+1}[x] ← key_i[x]
                i ← i − 1
         key_{i+1}[x] ← k
         n[x] ← n[x] + 1
         DISK-WRITE(x)
    else while i ≥ 1 and k < key_i[x]
             do i ← i − 1
         i ← i + 1
         DISK-READ(c_i[x])
         if n[c_i[x]] = 2t − 1
           then B-TREE-SPLIT-CHILD(x, i, c_i[x])
                if k > key_i[x]
                  then i ← i + 1
         B-TREE-INSERT-NONFULL(c_i[x], k)
```

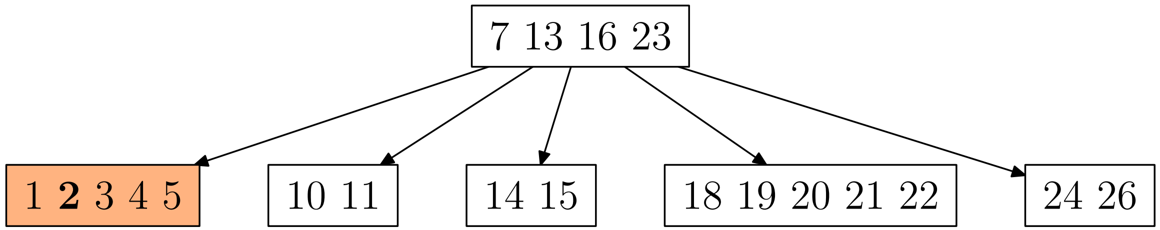
Inserting a key - Examples (I)

Initial tree:

$t = 3$

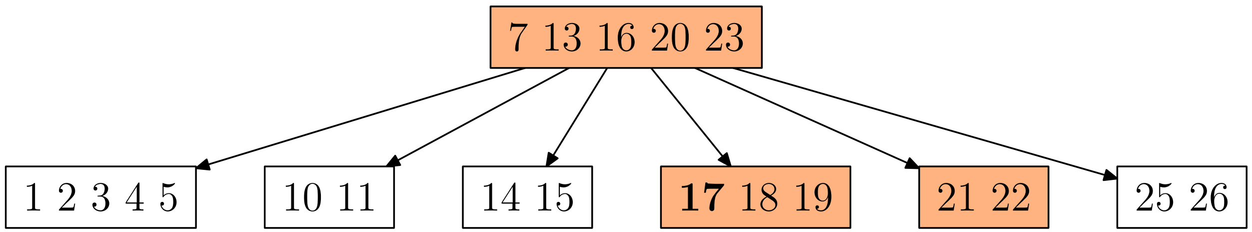


2 inserted:



17 inserted:

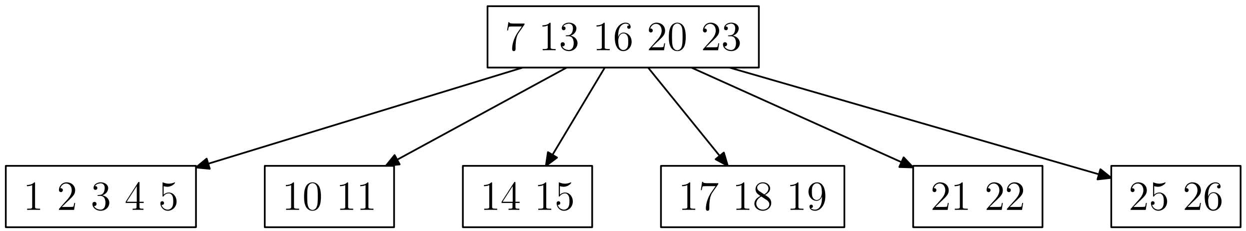
(to the previous one)



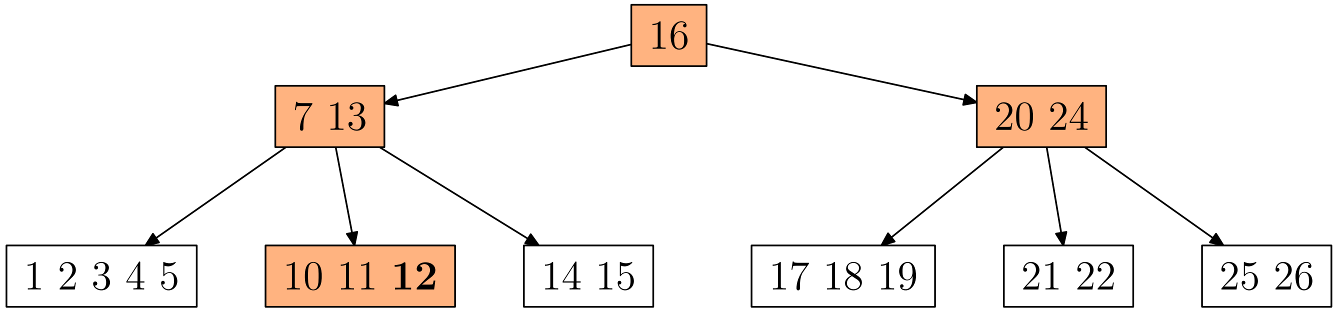
Inserting a key - Examples (II)

Initial tree:

$t = 3$

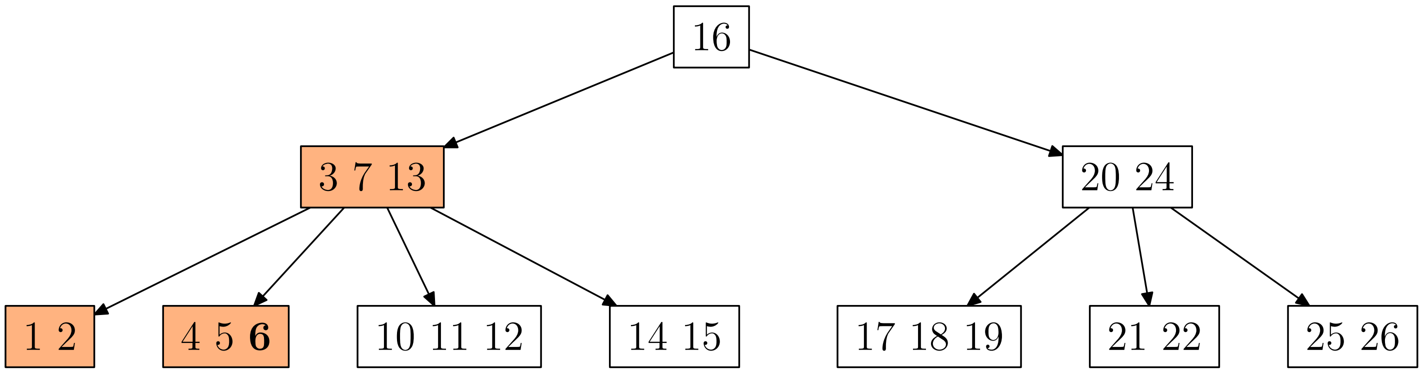


12 inserted:



6 inserted:

(to the previous one)



Deleting a Key from a B-tree

- Similar to insertion, with the addition of a couple of special cases
- Key can be deleted from any node.
- More complicated procedure, but similar performance figures: $O(h)$ disk accesses, $O(th) = O(t \log_t n)$ CPU time
- Deleting is done in a single pass down the tree, but needs to return to the node with the deleted key if it is an internal node
- In the latter case, the key is first moved down to a leaf. Final deletion **always** takes place on a **leaf**

Deleting a Key — Cases I

- Considering **3** distinct cases for deletion
- Let k be the key to be deleted, x the node containing the key. Then the cases are:

1. If key k is in node x and x is a leaf, simply delete k from x

2. If key k is in node x and x is an internal node, there are three cases to consider:

(a) If the child y that precedes k in node x has at least t keys (more than the minimum), then find the predecessor key k' in the subtree rooted at y. Recursively delete k' and replace k with k' in x

(b) Symmetrically, if the child z that follows k in node x has at least t keys, find the successor k' and delete and replace as before. Note that finding k' and deleting it can be performed in a single downward pass

(c) Otherwise, if both y and z have only $t-1$ (minimum number) keys, merge k and all of z into y, so that both k and the pointer to z are removed from x. y now contains $2t-1$ keys, and subsequently k is deleted

Deleting a Key — Cases II

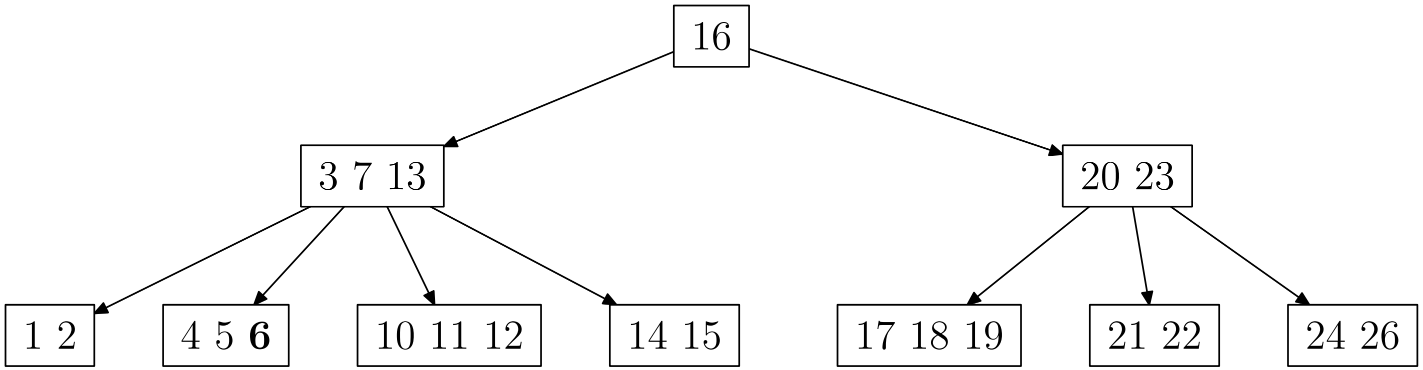
3. If key k is not present in an internal node x, determine the root of the appropriate subtree that must contain k. If the root has only $t-1$ keys, execute either of the following two cases to ensure that we descend to a node containing **at least** t keys. Finally, recurse to the appropriate child of x

(a) If the root has only $t-1$ keys but has a sibling with t keys, give the root an extra key by moving a key from x to the root, moving a key from the roots immediate left or right sibling up into x, and moving the appropriate child from the sibling to x

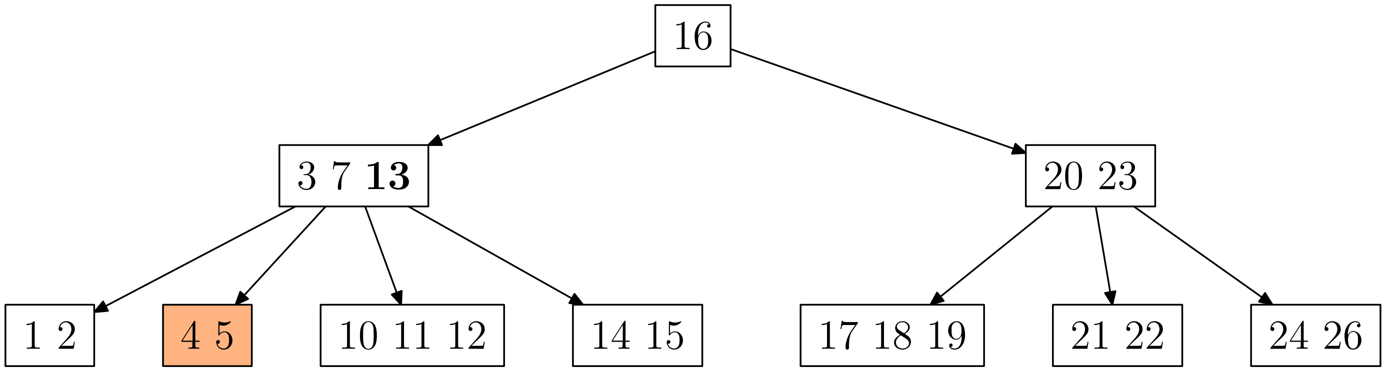
(b) If the root and all of its siblings have $t-1$ keys, merge the root with one sibling. This involves moving a key down from x into the new merged node to become the median key for that node.

Deleting a Key — Case 1

Initial tree:



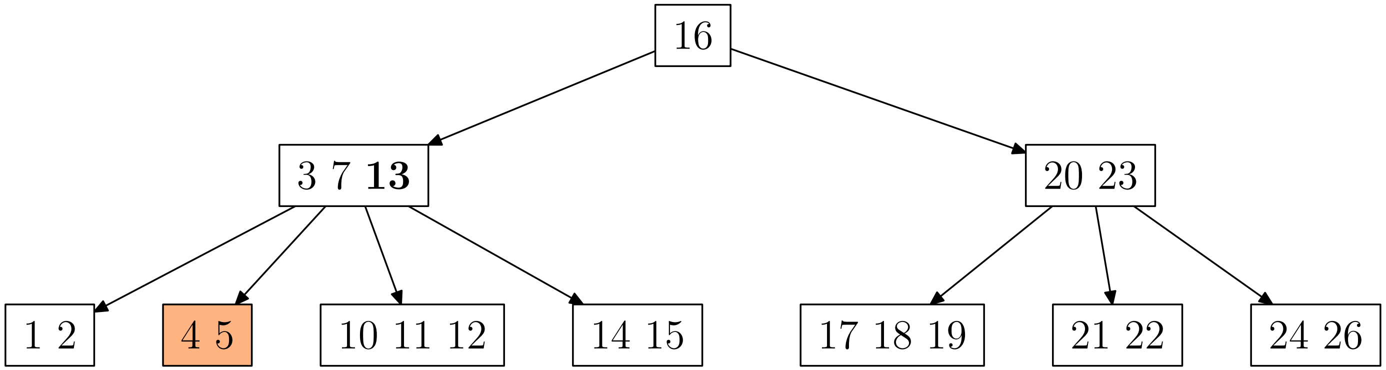
6 deleted:



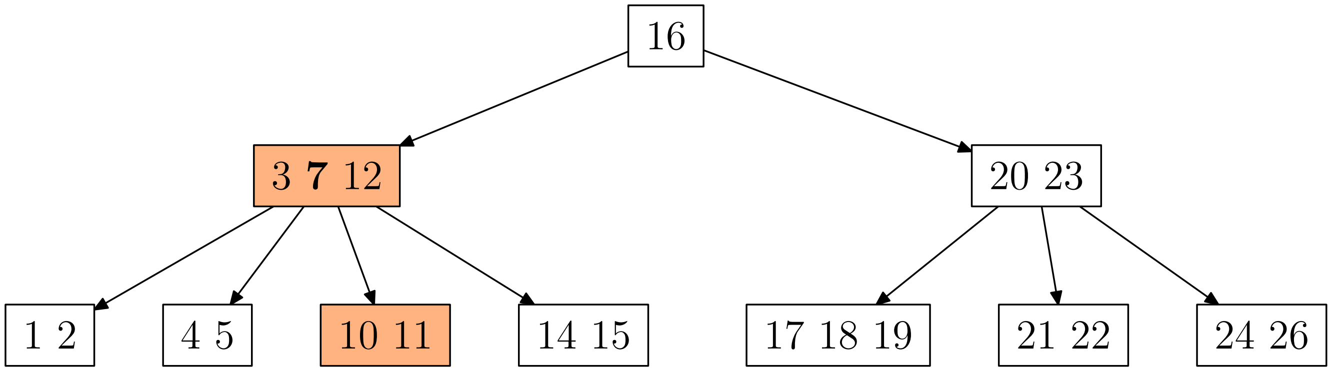
- The first and simple case involves deleting the key from the leaf. $t - 1$ keys remain

Deleting a Key — Cases 2a, 2b

Initial tree:



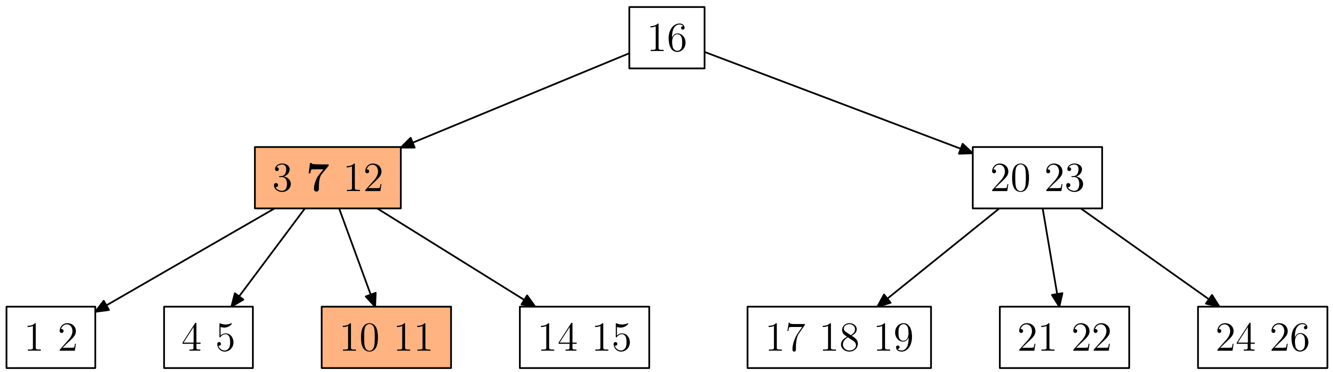
13 deleted:



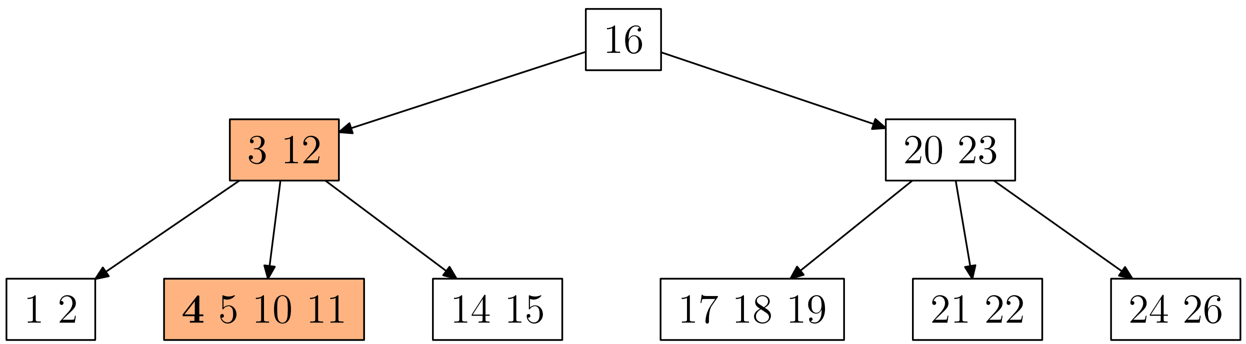
- Case $2a$ is illustrated. The predecessor of 13, which lies in the preceding child of x, is moved up and takes 13s position. The preceding child had a key to spare in this case

Deleting a Key — Case 2c

Initial tree:



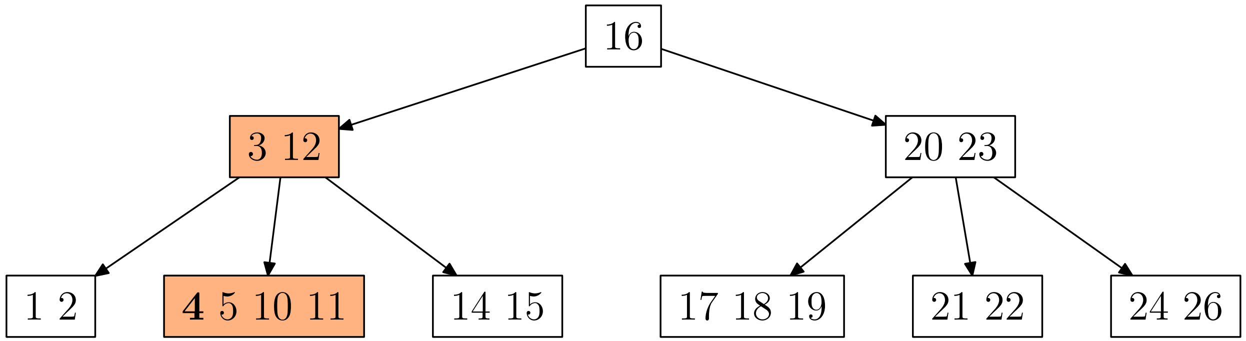
7 deleted:



- Here, both the preceding and successor children have $t-1$ keys, the minimum allowed. 7 is initially pushed down and between the children nodes to form one leaf, and is subsequently removed from that leaf

Deleting a Key — Case 3b

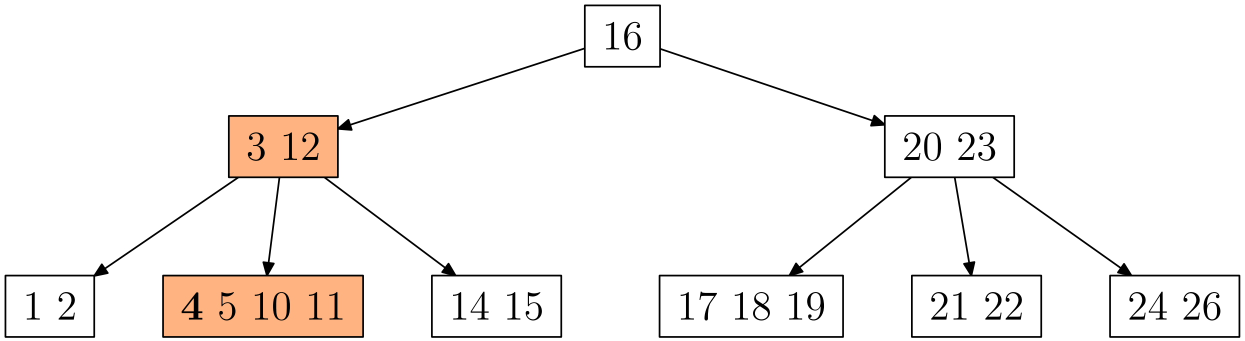
Initial tree: Key **4** to be deleted



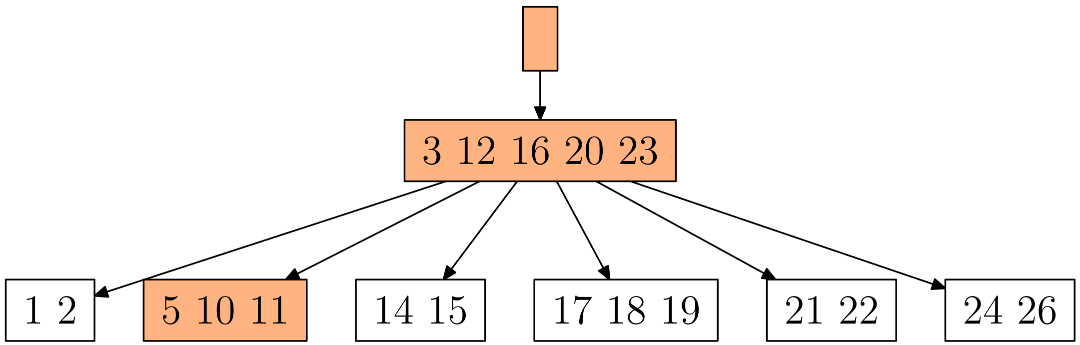
- The catchy part. Recursion cannot descend to node $3, 12$ because it has $t-1$ keys. In case the two leaves to the left and right had more than $t-1$, $3, 12$ could take one and 3 would be moved down.
- Also, the sibling of $3, 12$ has also $t-1$ keys, so it is not possible to move the root to the left and take the leftmost key from the sibling to be the new root
- Therefore the root has to be pushed down merging its two children, so that 4 can be safely deleted from the leaf

Deleting a Key — Case 3b II

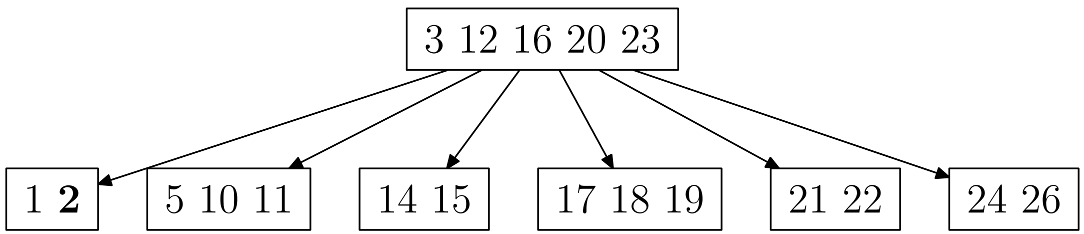
Initial tree:



4 deleted:

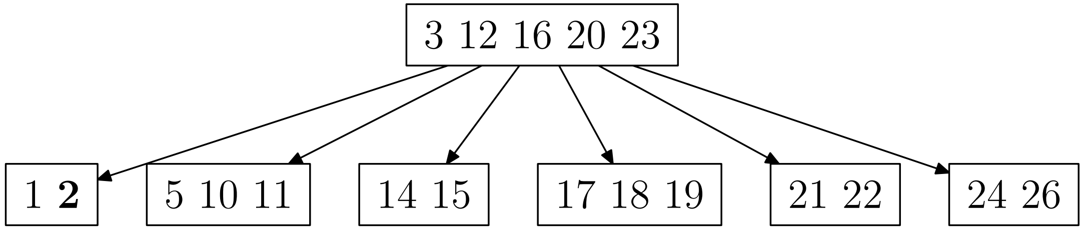


Outcome:



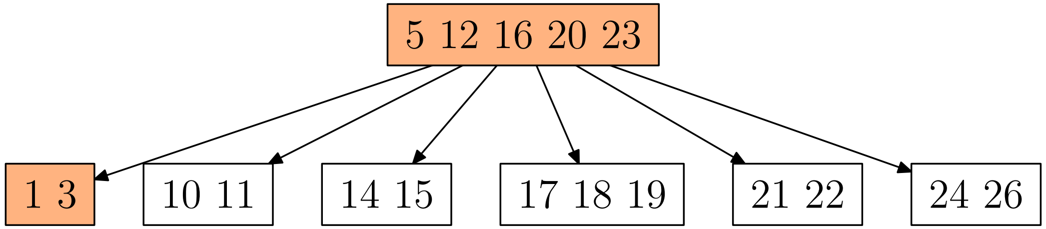
Deleting a Key — Case 3a

Initial tree:



2 deleted:

(to the previous one)



- In this case, $1, 2$ has $t-1$ keys, but the sibling to the right has t. Recursion moves 5 to fill 3s position, 5 is moved to the appropriate leaf, and deleted from there

Deleting a Key — Pseudo Code I

```
B-TREE-DELETE-KEY(x, k)
  if not leaf[x] then
    y ← PRECEDING-CHILD(x)
    z ← SUCCESSOR-CHILD(x)
    if n[y] > t − 1 then
        k′ ← FIND-PREDECESSOR-KEY(k, x)
        MOVE-KEY(k′, y, x)
        MOVE-KEY(k, x, z)
        B-TREE-DELETE-KEY(k, z)
    else if n[z] > t − 1 then
        k′ ← FIND-SUCCESSOR-KEY(k, x)
        MOVE-KEY(k′, z, x)
        MOVE-KEY(k, x, y)
        B-TREE-DELETE-KEY(k, y)
    else
        MOVE-KEY(k, x, y)
        MERGE-NODES(y, z)
        B-TREE-DELETE-KEY(k, y)
```

Deleting a Key — Pseudo Code II

```
else (leaf node)
  y ← PRECEDING-CHILD(x)
  z ← SUCCESSOR-CHILD(x)
  w ← root(x)
  v ← RootKey(x)
    if n[x] > t − 1 then REMOVE-KEY(k, x)
    else if n[y] > t − 1 then
        k' ← FIND-PREDECESSOR-KEY(w, v)
        MOVE-KEY(k', y, w)
        k' ← FIND-SUCCESSOR-KEY(w, v)
        MOVE-KEY(k', w, x)
        B-TREE-DELETE-KEY(k, x)
    else if n[w] > t − 1 then
        k' ← FIND-SUCCESSOR-KEY(w, v)
        MOVE-KEY(k', z, w)
        k' ← FIND-PREDECESSOR-KEY(w, v)
        MOVE-KEY(k', w, x)
        B-TREE-DELETE-KEY(k, x)
```

Deleting a Key — Pseudo Code III

```
else
    s ← Find-Sibling(w)
    w′ ← root(w)
        if n[w′] = t − 1 then
            Merge-Nodes(w′, w)
            Merge-Nodes(w, s)
            B-Tree-Delete-Key(k, x)
        else
            Move-Key(v, w, x)
            B-Tree-Delete-Key(k, x)
```

- $\textsc{Preceding-Child}(x)$ Returns the left child of key x.
- $\textsc{Move-Key}(k, n_1, n_2)$ Moves key k from node n_1 to node n_2.
- $\textsc{Merge-Nodes}(n_1, n_2)$ Merges the keys of nodes n_1 and n_2 into a new node.
- $\textsc{Find-Predecessor-Key}(n, k)$ Returns the key preceding key k in the child of node n.
- $\textsc{Remove-Key}(k, n)$ Deletes key k from node n. n must be a leaf node.